OpenCL vs OpenACC: Lessons from Development of Lattice QCD Simulation Code

AUTHOR(S):

Matsufuru, H.; Aoki, S.; Aoyama, T.; Kanaya, K.; Motoki, S.; Namekawa, Y.; Nemura, H.; Taniguchi, Y.; Ueda, S.; Ukita, N.

CITATION:

Matsufuru, H. ...[et al]. OpenCL vs OpenACC: Lessons from Development of Lattice QCD Simulation Code. Procedia Computer Science 2015, 51: 1313-1322

ISSUE DATE:

2015

URL:

http://hdl.handle.net/2433/226402

Procedia Computer Science

Volume 51, 2015, Pages 1313–1322

ICCS 2015 International Conference On Computational Science

OpenCL vs OpenACC: lessons from development of lattice QCD simulation code

H. Matsufuru[1], S. Aoki[2], T. Aoyama[3], K. Kanaya[4], S. Motoki[5], Y. Namekawa[6], H. Nemura[6], Y. Taniguchi[4], S. Ueda[7], and N. Ukita[6] (Bridge++ Project)

[1] Computing Research Center, High Energy Accelerator Research Organization (KEK), Japan, and Graduate University for Advanced Studies (Sokendai), Japan.
hideo.matsufuru@kek.jp
[2] Yukawa Institute for Theoretical Physics, Kyoto University, Japan.
saoki@yukawa.kyoto-u.ac.jp
[3] Kobayashi-Maskawa Institute for the Origin of Particles and the Universe (KMI), Nagoya University, Japan. aoym@kmi.nagoya-u.ac.jp
[4] Graduate School of Pure and Applied Sciences, University of Tsukuba, Japan.
{kanaya, tanigchi}@ccs.tsukuba.ac.jp
[5] Computing Research Center, High Energy Accelerator Research Organization (KEK), Japan.
smotoki@post.kek.jp
[6] Center for Computational Sciences, University of Tsukuba, Japan.
{namekawa, nemura, ukita}@ccs.tsukuba.ac.jp
[7] Theory Center, IPNS, High Energy Accelerator Research Organization (KEK), Japan.
sueda@post.kek.jp

Abstract

OpenCL and OpenACC are generic frameworks for heterogeneous programming using CPU and accelerator devices such as GPUs. They have contrasting features: the former explicitly controls devices through API functions, while the latter generates such procedures along a guide of the directives inserted by a programmer. In this paper, we apply these two frameworks to a general-purpose code set for numerical simulations of lattice QCD, which is a computational physics of elementary particles based on the Monte Carlo method. The fermion matrix inversion, which is usually the most time-consuming part of the lattice QCD simulations, is offloaded to the accelerator devices. From a viewpoint of constructing reusable components based on the object-oriented programming and also tuning the code to achieve high performance, we discuss feasibility of these frameworks through the practical implementations.

Keywords: Lattice gauge theory, Accelerator, OpenCL, OpenACC

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2015

doi:10.1016/j.procs.2015.05.316

1 Introduction

Rapid increase of computer resources has made numerical simulations powerful tools in many fields of science. In elementary particle physics, Quantum Chromodynamics (QCD), which describes dynamics of interaction among quarks, is a prominent example. Since an analytic method is generally not applicable to low energy physics of QCD, numerical simulations have been only the way of quantitative calculation for many quantities. Numerical simulations of QCD are based on the quantum field theory on the discretized Euclidean spacetime (lattice). This so-called lattice QCD enables numerical computation of quantum expectation values by evaluating the path integral using a Monte Carlo method [11, 5]. With recent large computational power, the lattice QCD produces more and more precise results, and furthermore it is extensively applied to other field theories, *e.g.* in search for a theory beyond the standard model of particle physics.

Nowadays, high performance computing device has become popular. In addition to parallel clusters, accelerator devices such as GPUs and Xeon Phi provides large computational power with relatively low cost. It expedites application to variety of calculations. On the other hand, development of simulation code has become more and more involved. One generally needs to write a hybrid parallel code for multi-node and multi-thread machines. Furthermore, the code must be modified to offload tasks of hot spots to accelerator devices. To apply to wide range of physical models and numerical algorithms as well as hardware architecture, a code should separate its ingredients with least interference. For example, the code specific to each hardware must be encapsulated into small part of the program, and can be combined with any models and algorithms.

A guideline to develop such a program is the object-oriented programming (OOP). In 2009, we launched a project to develop a code set for lattice QCD simulations that is widely applicable while based on uniform design policy guided by OOP [2, 13]. The code set, named Bridge++, is written in C++. This paper concerns the design of Bridge++ to incorporate the accelerator devices. There are several possibilities in a choice of frameworks for accelerator devices, such as CUDA, OpenCL, and OpenACC. Since, as discussed later, one of the goals of Bridge++ is portability, the latter two are our feasible candidates. OpenCL and OpenACC is based on different policy. OpenCL explicitly controls the devices through API functions. OpenACC is a directive-based extension of programming language. In this paper, we apply OpenCL and OpenACC to Bridge++ for offloading the linear equation solver to the accelerator devices. From a point of view of constructing reusable components based on OOP, and also tuning the code to achieve high performance, we evaluate feasibility of these two frameworks through the practical implementations.

This paper is organized as follows. Section 2 explains a hot spot of the numerical simulation of lattice QCD that is to be executed on the device. Section 3 introduces our code set Bridge++. The way of our implementation in a context of the object-oriented construction is described. Section 4 presents a strategy for offloading tasks of lattice simulation to accelerators. Section 5 and Section 6 describe our implementation in OpenCL and OpenACC, respectively. In Section 7, after summarizing our test environment, performance of these codes are reported. Section 8 is devoted to discussion and conclusion.

2 Lattice QCD simulations

For the formulation of lattice QCD and a principle of the numerical simulation, there are many textbooks and reviews [11, 5]. Thus we concentrate on solving a linear equation for a fermion

matrix, which is in many cases the most time consuming part in the numerical simulation.

A lattice QCD action consists of fermion (quark) fields and a gauge (gluon) field. The latter mediates interaction between quarks and are represented by link variables, $U_\mu(x) \in SU(N_c)$ ($N_c = 3$), where $x = (x_1, x_2, x_3, x_4)$ stands for a lattice site and μ=1–4 is a spacetime direction. In numerical simulations the size of a lattice is finite, $x_\mu = 1, 2, \ldots, L_\mu$. The fermion field is represented as a complex vector on lattice sites, which carries 3 components of color and 4 components of spinor. The fermion action is a bilinear of fermion field ψ, $S_F = \sum_{x,y} \bar{\psi}(x) D[U](x,y) \psi(y)$, where $D[U]$ is a fermion operator. A Monte Carlo algorithm is applied to generate an ensemble of the gauge field $\{U_\mu(x)\}$, that requires to solve a linear equation $v = D^{-1}b$ many times.

There are several fermion formulations. Each formulation has its own advantages and disadvantages. They may be combined with a variety of improvement procedures. As a common feature, a fermion operator is represented with local interactions, so that the fermion matrix is sparse. As a representative, we here consider the Wilson fermion operator,

$$D_W(x,y) = \delta_{x,y} - \kappa \sum_{\mu=1}^{d} \left[(1-\gamma_\mu) U_\mu(x) \delta_{x+\hat{\mu},y} + (1+\gamma_\mu) U_\mu^\dagger(x-\hat{\mu}) \delta_{x-\hat{\mu},y} \right] \tag{1}$$

where x, y are lattice sites, $d = 4$ is the space-time dimension, $\hat{\mu}$ is a unit vector along the μ-th axis. γ_μ is 4×4 matrix acting on the spin degree of freedom. κ is a parameter related to the fermion mass. Thus D_W is a $4N_cL_xL_yL_zL_t$ dimensional complex matrix. The Wilson fermion has several improved variants so that lattice artifact is decreased. Although the Wilson-type operators are extensively used, it has a disadvantage that it explicitly breaks the chiral symmetry. The chiral symmetry holds in the continuum limit for a massless fermion and plays an important role in the dynamics of QCD. There are other fermion formulations that hold the chiral symmetry better or exactly at the expense of higher computational cost.

Similarly, the linear equation solver algorithms also have variety. Since the fermion operator is represented as a sparse matrix, iterative solver based on the Krylov subspace method is in general used. According to the properties of the matrix, there are a number of algorithms available, such as the conjugate gradient (CG) for a hermitian positive definite matrix and bi-conjugate gradient (BiCG) for a nonhermitian matrix. Although for the Wilson fermion operator above, the BiCGStab algorithm or its variant is usually efficient, other algorithm might be better for other fermion formulation. In addition, for large-scale linear systems, there are variety of improvement techniques, such as a multi-grid method. When the memory bandwidth is a bottleneck, which is frequently the case for GPUs, a single precision solver is often employed as a preconditioner that brings considerable speed up.

These situations indicate that general-purpose code set should be able to exchange the fermion matrices and solver algorithms independently.

3 An object-oriented C++ code Bridge++

This paper aims at comparing OpenCL and OpenACC in applying to a general purpose code to offload tasks to accelerator devices. To clarify the required conditions and viewpoint of comparison, we first describe our base code named Bridge++ [2]. Bridge++ is intended to possess the following features:

- Readability: the code structure is transparent so as to be understandable even for beginners.
- Extensibility: the code is easy to be modified for testing new ideas.

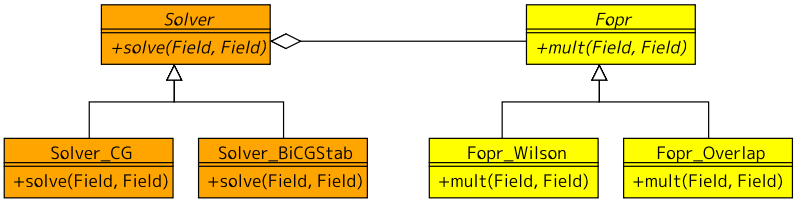

Figure 1: A class diagram of solver and fermion operator which shows their relation as an application of the bridge pattern.

- Portability: the code runs on wide range of architectures, from laptop PCs to supercomputers.
- High-performance: the code has sufficiently high performance for productive research.

To achieve these goals simultaneously, we make use of the insight of object-oriented programming (OOP) by describing the code in C++ programming language. Bridge++ is parallelized by MPI for nodes with distributed memory [13]. In the latest version 1.2, we started to support the hybrid parallelization employing OpenMP as a multi-threading library.

As noted in the previous section, how to incorporate the variety of operators and algorithms is a subject of the code design. This is unraveled by so-called 'Bridge pattern', one of GoF's design patterns [9] which is common wisdom for reuse of the code. Figure 1 displays the class diagram of our current design. The abstract classes *Solver* and *Fopr* define the interfaces of linear equation solvers and fermion operators, respectively. Each solver algorithm is implemented as a subclass of *Solver* class, while having a member function `solve()` as a common interface. Similarly, each fermion operator is implemented as a subclass of *Fopr* whose common interface `mult()`, matrix multiplication to a vector, is called from a solver. (The overlap operator is one of fermion formulations.) Thus solver and the fermion operator are implemented separately. While such structure is of course made up without OOP, OOP plays a guide of finding good design and may provide common terminology such as design patterns. The implementation design of Bridge++ indeed significantly owes the design patterns.

The fermion matrix acts on a vector whose degree of freedom is $4N_cL_xL_yL_zL_t$. When one parallelizes the system, this vector lies over multi-nodes and multi-threads. In present implementation of Bridge++, such a vector is represented as an object of the `Field` class. It encapsulates the actual data structure, though practically a contiguous array of double precision floating-point data is assumed. For the `Field` class, operations corresponding to BLAS are defined, in which inter-node and multi-thread operations are encapsulated, so that the linear algebra is constructed using these operations.

In developing the code for offloading linear fermion solvers to accelerators, we extend the above implementation. Although the `Field` class assumes double precision data, it is frequently convenient to change the precision of data, for example in applying multi-precision linear solvers. Thus we implement alternative field container, `AField<REALTYPE>`, where `REALTYPE` is a data type, float or double. Corresponding solvers and fermion operators are also implemented using C++ template.

4 Strategy to use accelerator devices

The performance of arithmetic accelerators such as GPGPUs are rapidly being improved. These devices provide a large computational power with less cost and electricity. They have already

been widely used in lattice simulations [8, 3]. There are indeed open source libraries such as QUDA [4] for NVIDIA GPUs which is a CUDA-based library for lattice QCD. According to our aims, however, we develop a code for general accelerator devices so as to establish techniques to fully make use of their performance. In addition to CUDA SDK for NVIDIA's GPUs, there are several programming frameworks. Among them, OpenCL and OpenACC are attracting candidates because they can be applied to a wide range of architecture. They have contrasting features: the former is API-based which controls the devices explicitly, like Pthread for multi-threading, while the latter is directive-based and a compiler generates procedures that use devices, like OpenMP. Application of OpenCL [1, 6, 7, 12] and OpenACC [10] to lattice QCD have recently been started.

To incorporate a code to offload the tasks of lattice simulations into Bridge++, we require that the following conditions are satisfied.

- Explicit calls of procedures to control the accelerator devices are encapsulated in a small number of classes.
- Single and double (and perhaps other types of) precision can be treated simultaneously.
- Parameters such as lattice sizes can be changed at run-time.
- Performance is acceptably high with a small effort of tuning.

When one uses accelerator devices, the following steps are necessary to be executed.

(1) Get information of accelerators and setup environment.
(2) Setup kernel code.
(3) Allocate memory space for data on device.
(4) Transfer data from host to device.
(5) Execute kernel code.
(6) Transfer data from device to host.
(7) Free the memory space on device.

Usually steps (4) and (6) become bottlenecks, because of a narrow bandwidth between the host and device. Thus the data transfer between host and device should be minimized as much as possible to achieve better performance.

To represent the data on the device, we define a class `AField_dev<REALTYPE>` that corresponds to the `AField<REALTYPE>` class on the host. The `AField_dev` class contains member functions that handle the data transfer between the host and device. At the construction of `AField_dev`, the memory space on the device is allocated that is freed at the destruction. Linear algebraic operations corresponding to the BLAS routines are also prepared for instances of `AField_dev`. Thus for allocation, data transfer, vector operations, and deallocation of data are encapsulated. Corresponding linear solver algorithms and fermion operators are defined so as to operate on instances of `AField_dev` class.

As common implementation to OpenCL and OpenACC, operations on each lattice site are assigned to one thread. To optimize the data transfer between global memory and cores, we apply so-called coalesced memory access by changing the data layout on the device from that on the host. To reduce memory transfer, the third column of SU(3) matrix in gauge field is not transferred from the global memory but calculated on-the-fly using the relation $v_3 = (v_1 \times v_2)^*$ where $U = (v_1, v_2, v_3) \in SU(3)$. The sustained performance demonstrated below does not include the operations for this reconstruction. Even with this implementation, if the arithmetic operations are well optimized, the memory transfer is the bottleneck and determines the practical performance.

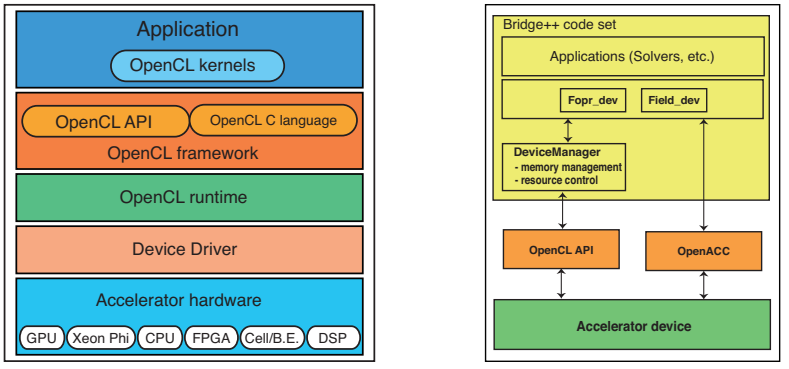

Figure 2: Left panel: schematic structure of OpenCL framework. Right panel: schematic structure of Bridge++ to handle the accelerator.

5 Implementation with OpenCL

5.1 OpenCL

OpenCL (Open Computing Language) is an open standard framework for a parallel programming in heterogeneous platforms, such as CPUs, GPUs, FPGAs, and other types of processors. The specifications are maintained by Khronos Group with contributions of hardware and software companies. The left panel of Figure 2 shows an image of OpenCL framework. OpenCL works on an abstract hardware layer (orange part) and controls accelerator hardware (blue part) through it. Thus applications can be developed independently of specific architecture. This matches our design policy with respect to portability. Nonetheless, one needs to understand the hardware structure to fully extract the potential performance of the device.

The specification of OpenCL is composed of the run-time APIs and the OpenCL C language. The former is used to control the devices from the host programs, and the latter is prepared for describing the device codes. On devices, threads run in parallel executing the same program. A thread is called *work-item*, and a specified number of work-items are grouped to form a *work-group*. Device memory is classified into four types.

- Global memory: readable and writable from all the work-items and from the host.
- Constant memory: read-only from all work-items, readable and writable from the host.
- Local memory: shared by work-items within a work-group.
- Private memory: exclusively used by a work-item.

The total number of work-items and size of work-group are tunable parameters at run-time.

5.2 Implementation in Bridge++

Each step of the work flow in Sec. 4 can be handled through OpenCL APIs. At the initialization step (1), one needs to obtain the information of platforms, setup *contexts* and *command queues* at the beginning of a program. If a run-time compiler is used, as we adopt, a kernel code is compiled at step (2). To avoid explicit appearance of these OpenCL APIs in individual classes, we develop a `DeviceManager` class that encapsulates the OpenCL APIs, so as to simplify the procedures and to switch the framework easily. The right panel of Figure 2 schematically expresses the adopted design. This `DeviceManager` class also wraps management of device

memory objects and data transfer between the host and devices. Each object accesses to the device memory through the interfaces provided by the `DeviceManager` class.

The device code, described in OpenCL C language, is embedded as a string object at the compilation of Bridge++ code, and then on-line compiled at the run-time. Using this mechanism, several parameters and functions described as macros are replaced at the run-time. It is expected that this helps optimization. Changing the data layout is possible in this way, by replacing the macro definition of an index, without modifying the kernel code.

The field data on device memory is handled as an object of the `AField_dev<REALTYPE>` class. At the construction of an instance of this class, the associated memory space on the device is allocated through the `DeviceManager` class (step (4) of the workflow), and it is released at the destruction (step (7)). This class contains methods to transfer data between the host and the device (steps (4) and (6)). The linear-algebraic operations in analogous to the BLAS routines (for step (5)) are prepared as methods, and the kernel codes used in these methods are compiled and cached through the `DeviceManager` when the first instance is constructed. Using this class, for example, the solver algorithms can be constructed in a general manner. The fermion operators on device, represented as `Fopr_dev` class in the right panel of Fig. 2), are implemented similarly. By using these objects, applications, such as a linear equation solver, are offloaded to the accelerator device.

6 Implementation with OpenACC

6.1 OpenACC

OpenACC is a directive-based extension of languages. The standard is defined for C/C++ and Fortran. A user inserts directives to the code. Then a compiler analyzes them and generates the procedures for offloading data and tasks to accelerators. Currently the standard 2.0 is available.

OpenACC assumes 3 levels in the processor: *gang*, *worker*, and *vector*. For example in NVIDIA Tesla architecture, they respectively correspond to the streaming multi-processor, warp, and thread. For C/C++, a directive is inserted to a code as pragma with a general syntax of

```
#pragma acc directive-name [clauses]
```

Three kinds of directives are important.

- Specification of parallel region:
  Two directives, `kernels` and `parallel`, are defined in OpenACC to specify which part of the code is to be executed in parallel. While the `kernels` directive entrusts a compiler with responsibility in analyzing dependencies of variables, the `parallel` directive implies that owe to the user. We use the latter in our implementation.
- Memory allocation and data transfer:
  `data` directive is a representative example. From OpenACC 2.0, `enter` and `exit` directives are added which allocate and free a memory space on the device. Data transfer between host and device is executed by `update` directive. Then before the parallel region, the clause of `data` directive is always `present`.
- Specification of parallelized loop:
  This is done by `loop` directive. In `parallel` region, this is necessary to be specified. With a clause, one can specify which of gang, worker, and vector is assigned to the loop, and variables that are private to the loop. Collapsing loops and specification of reduction can also be indicated with clauses.

6.2 Implementation in Bridge++

Since the OpenACC libraries that control the devices are implicitly called from a code by insertion of directives, we implement no class that corresponds to the `DeviceManager` class for OpenCL. Instead we directly modify the code that uses devices by inserting OpenACC directives (Cf. the right panel of Fig. 2). The steps (1) and (2) of the work flow in Section 4 are automatically incorporated by the compiler. As in the case of OpenCL, we define `AField_dev<REALTYPE>` class that represents field data on the device. At the construction of an instance of this class, the constructor allocates memory space of the device by `enter` directive (step (3)). This device memory space is freed in the destructor by `exit` directive (step(7)). To transfer data between the host and device, member functions are defined using the `update` directive (steps (4) and (6)). This implementation enables explicit control of memory allocation and data transfer through an abstract interface.

The kernel code to execute the step (5) is generated by a compiler at the `parallel` directive. In the `AField_dev<REALTYPE>` class, BLAS methods are implemented in this manner. Since the data transfer is managed in a separated class by using `update` directive, the clause of data directive before the parallel directive is in general `present` which indicates that the memory space has already been allocated and the data is ready. The fermion matrix multiplication, the `Fopr_dev` class in Fig. 2, is implemented in the same way. By replacing corresponding objects in the OpenCL version with them, the solver algorithm works without modification.

7 Performance

In this section, we report the sustained performance obtained for implementation within Bridge++ using OpenCL and OpenACC. First we summarize our test environment in which the following two types of accelerators are tested.

NVIDIA Tesla K40 (Kepler architecture)

- Peak performance: 4290 GFlops (float), 1430 GFlops (double)
- Global memory bandwidth: 288 GB/s
- Number of cores: 2880, 192 cores/streaming multi-processor
- CUDA 5.5
- PGI compiler 14.10 (OpenACC)

AMD Radeon HD7970 (Tahiti architecture)

- Peak performance: 3789 GFlops (float), 947 GFlops (double)
- Global memory bandwidth: 264 GB/s
- Number of cores: 2048, 64 cores/wavefront
- AMD APP SDK v.2.9

As a representative example, we implement the Wilson fermion operator defined as in Eq. (1) and the conjugate gradient (CG) solver algorithm. The performance is measured for a multiplication of the Wilson operator (represented as "mult") and for the CG solver on a $16^3 \times 32$ lattice. In the above, we use single GPU device. While we have no OpenACC environment on our host with AMD GPU, we include the result for OpenCL in order to examine the portability of our code.

	OpenCL		OpenACC	
operation	float	double	float	double
NVIDIA Tesla K40:				
Wilson mult	235 GFlops	121 GFlops	202 GFlops	38.4 GFlops
CG solver	161 GFlops	86.2 GFlops	126 GFlops	34.6 GFlops
AMD Radeon HD7970:				
Wilson mult	228 GFlops	110 GFlops	N/A	N/A
CG solver	73.1 GFlops	47.7 GFlops	N/A	N/A

Table 1: Performance with OpenCL and OpenACC for Wilson fermion matrix mult and CG solver on a $16^3 \times 32$ lattice.

Performance with OpenCL In Table 1, we summarize the present performance of our code on NVIDIA and AMD GPUs. This is an update of the result reported in Ref. [12]. The run-time parameters for thread grouping are adjusted for each device. The results of the single precision almost twice the double precision indicate that the performance is indeed determined by the data transfer between the device memory and cores. While the performance of the Wilson matrix multiplication is comparable for NVIDIA and AMD GPUs, the performance of the solver shows amplified differences. This is presumably caused by inefficient implementation of the reduction in the inner-product. At present, the BLAS methods are implemented in the Bridge++ code. We apply two step reduction, first reducing to a coarse-grained array and then taking a full reduction. It is found that these reductions affect the performance significantly. Further tuning of the code and architecture dependent optimization are underway. Adopting public library with better implementation may improve the performance of the linear algebraic part so as to accelerate the solver algorithm.

Performance with OpenACC In Table 1, we quote the present performance of OpenACC implementation of Bridge++. At present the result is available only for the NVIDIA GPU. The fermion multiplication is less efficient than the OpenCL version, in particular for the double precision version. The latter may due to non-optimal assignment of variables to registers. By reducing the number of local variables, the performance indeed approaches the half the values of the float cases. More careful tuning including modification of code may be necessary to achieve an improved performance.

8 Discussion and conclusion

We summarize advantages and disadvantages of OpenCL and OpenACC based on our experience of the implementation to lattice QCD code set Bridge++.

OpenCL requires a complicated setup, such as preparation of contexts, command queues, as well as a compilation of kernel codes. These setups are, however, not an obstacle of further development, once they are encapsulated in a management class. OpenCL API's provide the way to control devices in detail. While C++ template programming is not available in current OpenCL C language, on-line compiler provides alternative way to achieve polymorphism. It is also convenient in tuning the code that the memory type of a variable can be specified explicitly.

OpenACC has contrasting features to OpenCL. It is particularly effective at a start-up of the implementation. Once entering tuning phase, however, one needs to control the behavior of the compiler indirectly. This has made us spend more time to tune the OpenACC version

than OpenCL. Since the OpenACC compiler is rapidly becoming efficient, this tendency may be dissolved in the near future. We also note that the present OpenACC compiler we tested is not sufficiently mature in processing the C++ template syntax, which required us involved coding to enable static polymorphism.

Both frameworks allow us to offload the time consuming tasks keeping the object-oriented code structure. The interfaces implemented in Bridge++ are kept sufficiently simple so as to avoid potential overheads. Although the performance is better with OpenCL at present, that with OpenACC for single precision is acceptable. For the double precision case with OpenACC, more proper tuning is required. Nonetheless, both frameworks are expected to have rooms to improve their performance. Further tuning is ongoing, depending on each device architecture.

At this moment, we have not decided to select which of them as our base code. A practical solution seems to be to prepare these codes for accelerator devices as libraries to extend the core code set and to select one of them appropriately by considering the purpose and performance. Such code design is now under investigation.

Acknowledgments

The code was developed and tested on HA-PACS at University of Tsukuba under a support for its Interdisciplinary Computational Science Program, and workstations located at KEK Computing Research Center. This project is supported by Joint Institute for Computational Fundamental Science and HPCI Strategic Program Field 5 'The origin of matter and the universe'. This work is supported in part by JSPS Grant-in-Aid for Scientific Research (Nos. 20105005, 24540250, 25400284).

References

[1] M. Bach, V. Lindenstruth, O. Philipsen, and C. Pinke. Lattice QCD based on OpenCL. *Comput.Phys.Commun.*, 184:2042–2052, 2013.

[2] Bridge++ project. `http://bridge.kek.jp/Lattice-code/`. [online], 2012–2015.

[3] M.A. Clark. QCD on GPUs: cost effective supercomputing. *PoS*, LAT2009:003, 2009.

[4] M.A. Clark, R. Babich, K. Barros, R.C. Brower, and C. Rebbi. Solving Lattice QCD systems of equations using mixed precision solvers on GPUs. *Comput. Phys. Commun.*, 181:1517–1528, 2010.

[5] T. DeGrand and C. DeTar. *Lattice Methods for Quantum Chromodynamics.* World Scientific Pub., 2006.

[6] V. Demchik and N. Kolomoyets. QCDGPU: open-source package for Monte Carlo lattice simulations on OpenCL-compatible multi-GPU systems. *arXiv:1310.7087 [hep-lat]*, 2013.

[7] M. Di Pierro. QCL: OpenCL meta programming for lattice QCD. *PoS*, LATTICE2013:043, 2014.

[8] G. I. Egri, Z. Fodor, C. Hoelbling, S. D. Katz, D. Nogradi, and K. K. Szabo. Lattice QCD as a video game. *Comput.Phys.Commun.*, 177:631–639, 2007.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[10] P. Majumdar. Lattice Simulations using OpenACC compilers. *PoS*, LATTICE2013:031, 2014.

[11] I. Montvay and G. Münster. *Quantum Fields on a Lattice.* Cambridge Univ. Press, 1994.

[12] S. Motoki et al. Development of Lattice QCD Simulation Code Set on Accelerators. *Procedia Computer Science*, 29:1701, 2014.

[13] S. Ueda et al. Bridge++: an object-oriented C++ code for lattice simulations. *PoS*, LATTICE2013:412, 2014.